Merchant Documentation Documentation

Release 0.09a

Team Agiliq

August 27, 2015

Contents

1 Welcome to Merchant's documentation! 3

1.1 Welcome to Merchant's documentation! . . . 3
1.2 Indices and tables . . . 8998
1.3 Merchant: Pluggable and Unified API for Payment Processors . . . 8998
1.4 Overview . . . 8998
1.5 Installing Merchant . . . 8999
1.6 Credit Card . . . 9000
1.7 Gateways . . . 9002
1.8 On-site Processing . . . 9004
1.9 Authorize.Net Gateway . . . 9004
1.10 Beanstream . . . 9005
1.11 Bitcoin Gateway . . . 9006
1.12 Braintree Payments Server to Server . . . 9006
1.13 Chargebee . . . 9008
1.14 eWay Gateway . . . 9009
1.15 Paylane Gateway . . . 9010
1.16 PayPal Gateway . . . 9011
1.17 Stripe Payments . . . 9012
1.18 WePay Payments . . . 9013
1.19 Off-site Processing . . . 9015
1.20 PayPal Website Payments Standard . . . 9016
1.21 WorldPay . . . 9018
1.22 Amazon Flexible Payment Service . . . 9019
1.23 Braintree Payments Transparent Redirect . . . 9021
1.24 Stripe Payment Integration . . . 9023
1.25 eWAY Payment Integration . . . 9024
1.26 Authorize.Net Direct Post Method . . . 9027
1.27 Signals . . . 9029
1.28 Writing a new gateway . . . 9029
1.29 Customizing Merchant . . . 9030
1.30 Contributing to Merchant . . . 9031
1.31 Changes . . . 9031

2 Indices and tables 9035

3 Merchant: Pluggable and Unified API for Payment Processors 9037

4 Overview 9039

5 Installing Merchant **9041**
5.1 Post-installation . . . 9041
5.2 Configuration . . . 9041
5.3 Running the Test Suite . . . 9041

6 Credit Card **9043**
6.1 Attribute Reference . . . 9043
6.2 Method Reference . . . 9043
6.3 Subclasses . . . 9044

7 Gateways **9047**
7.1 Attribute Reference . . . 9047
7.2 Method Reference . . . 9047
7.3 Helper functions . . . 9049

8 On-site Processing **9051**

9 Authorize.Net Gateway **9053**
9.1 Usage . . . 9053

10 Beanstream **9055**
10.1 Example: . . . 9055

11 Bitcoin Gateway **9057**
11.1 Usage . . . 9057

12 Braintree Payments Server to Server **9059**
12.1 Example: . . . 9059

13 Chargebee **9061**
13.1 Example: . . . 9061

14 eWay Gateway **9063**
14.1 Usage . . . 9063

15 Paylane Gateway **9065**
15.1 Example: . . . 9065

16 PayPal Gateway **9067**
16.1 Usage . . . 9067

17 Stripe Payments **9069**
17.1 Example: . . . 9069

18 WePay Payments **9071**
18.1 Example: . . . 9071

19 Off-site Processing **9073**
19.1 Integration . . . 9073

20 PayPal Website Payments Standard **9075**
20.1 Test or Live Mode . . . 9075
20.2 Example . . . 9076

21 WorldPay **9079**
21.1 Example . . . 9079

22 Amazon Flexible Payment Service **9081**
22.1 Example . . . 9082

23 Braintree Payments Transparent Redirect **9085**
23.1 Example: . . . 9086

24 Stripe Payment Integration **9087**
24.1 Example: . . . 9087

25 eWAY Payment Integration **9089**
25.1 Example: . . . 9092

26 Authorize.Net Direct Post Method **9093**
26.1 Example: . . . 9094

27 Signals **9095**

28 Writing a new gateway **9097**

29 Customizing Merchant **9099**

30 Contributing to Merchant **9101**

31 Changes **9103**
31.1 0.4 (upcoming) . . . 9103
31.2 0.3 . . . 9103
31.3 0.2 . . . 9103
31.4 0.1 . . . 9103
31.5 0.09 . . . 9103
31.6 0.08 . . . 9103
31.7 0.07 . . . 9104
31.8 0.06 . . . 9104
31.9 0.05 . . . 9104
31.10 0.04 . . . 9104
31.11 0.03 . . . 9104
31.12 0.02 . . . 9104
31.13 0.01 . . . 9105

32 Indices and tables **9107**

Contents:

CHAPTER 1

Welcome to Merchant's documentation!

Contents:

1.1 Welcome to Merchant's documentation!

Contents:

1.1.1 Welcome to Merchant's documentation!

Contents:

Welcome to Merchant's documentation!

Contents:

Welcome to Merchant's documentation!

Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Welcome to Merchant's documentation! Contents:

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.

- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')

- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.

- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.

- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")
```

```
# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- MERCHANT_ACCOUNT_ID: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- PUBLIC_KEY: The public key provided by Braintree through their account dashboard.
- PRIVATE_KEY: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True
```

```
    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...
```

```
    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
```

```
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...
```

```
    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.

- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
```

```
    'pay_pal': {
        "WPP_USER" : '...',
        "WPP_PASSWORD" : '...',
        "WPP_SIGNATURE" : '...',
        "RECEIVER_EMAIL" : '...',
        # Below attribute is optional
        "ENCRYPTED": True
    }
    ...}

PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
```

```
...                                 {"obj": world_pay},
...                                 context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
```

```
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

check_transaction()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

check_transaction()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]\d)\d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]\d)\d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759\d{12}(\d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767\d{12}(\d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019\d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6\d)\d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722\d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)\d{8}(\d{4}|\d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = “Visa”
  - regexp = re.compile(‘^4d{12}(d{3})?$’)
- *MasterCard*
  - card_name = “MasterCard”
  - regexp = re.compile(‘^(5[1-5]d{4}|677189)d{10}$’)
- *Discover*
  - card_name = “Discover”
  - regexp = re.compile(‘^(6011|65d{2})d{12}$’)
- *AmericanExpress*
  - card_name = “Amex”
  - regexp = re.compile(‘^3[47]d{13}$’)
- *DinersClub*
  - card_name = “DinersClub”
  - regexp = re.compile(‘^3(0[0-5]|[68]d)d{11}$’)
- *JCB*
  - card_name = “JCB”
  - regexp = re.compile(‘^35(28|29|[3-8]d)d{12}$’)

Debit Cards

- *Switch*
  - card_name = “Switch”
  - regexp = re.compile(‘^6759d{12}(d{2,3})?$’)
- *Solo*
  - card_name = “Solo”
  - regexp = re.compile(‘^6767d{12}(d{2,3})?$’)
- *Dankort*
  - card_name = “Dankort”
  - regexp = re.compile(‘^5019d{12}$’)
- *Maestro*
  - card_name = “Maestro”
  - regexp = re.compile(‘^(5[06-8]|6d)d{10,17}$’)
- *Forbrugsforeningen*
  - card_name = “Forbrugsforeningen”
  - regexp = re.compile(‘^600722d{10}$’)

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

check_transaction()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
    - Signals
    - Writing your own gateway
    - Customizing the inbuilt Gateway/Integrations
    - Contributing to Merchant
    - Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]\d)\d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]\d)\d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759\d{12}(\d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767\d{12}(\d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019\d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6\d)\d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722\d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)\d{8}(\d{4}|\d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        - RBS WorldPay
        - Amazon FPS
        - Braintree Payments Transparent Redirect
        - Stripe
        - eWAY
        - Authorize.Net Direct Post Method
        - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
    (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
    (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
       (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.

- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
    pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**

- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
```

```
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)
```

```
    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)
```

```
# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's

IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
```

```
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
```

```
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
```

```
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
```

```
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information

- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`

- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
    def purchase(self, money, credit_card, options = None):
    ...

    def authorize(self, money, credit_card, options = None):
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
```

```
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.

- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"

  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.

- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.

- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)
```

```
    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
```

```
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
```

```
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- CLIENT_ID: This attribute refers to the application id that can be obtained from the account dashboard.
- CLIENT_SECRET: This is the secret for the corresponding CLIENT_ID.
- ACCOUNT_ID: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key account_id) in the views.
- ACCESS_TOKEN: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the settings.py file but has to be passed in the options dictionary (with the key token) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
```

```
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.

- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
```

```
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
```

```
            "paymentPage": request.build_absolute_uri(),
            # Send the correct url where the redirect should happen
            "returnURL": "%s://%s%s" % (url_scheme,
                                        domain,
                                        reverse("fps_return_url")),
        }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.

- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*

  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...
```

```
    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
    <integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
    modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
    class ModifiedBpIntegration(BraintreePaymentsIntegration):
        def braintree_success_handler(self, request, response):
            return render_to_response("my_new_success.html",
                                      {"resp": response},
                                      context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
    >>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change

- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * Global Iris
    - Off-site Processing
        * PayPal
        * Google Checkout
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * Global Iris RealMPI
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."
```

```
# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.

- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True          # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.

- LOGIN_COMPANY: The company name as visible from the account settings in the dashboard.
- LOGIN_USER: The username used to login to the account dashboard.
- LOGIN_PASSWORD: The password used to login to the account dashboard.
- HASH_ALGORITHM: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- HASHCODE: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
True

# A recurring plan charge
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        "recurring": {
            "plan_id": "test_plan",
            "trial_duration": 2,
            "trial_duration_unit": "month",
            "number_of_billing_cycles": 12,
            },
        }
>>> resp = braintree.recurring(10, credit_card, options = options)
>>> resp["response"].is_success
True
>>> resp["response"].subscription.number_of_billing_cycles
12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})
```

```
    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.

- SUDS_CACHE_DIR (optional): The location of the suds cache files. Defaults to /tmp/suds.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
```

```
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...
```

```
    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
```

```
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.

- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
```

```
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.

- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
```

```
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.

4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`

- `card_expiry_year`

`check_transaction`()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
`access_code`	
`authorisation_code`	`"198333"`
`response_code`	`"00"`
`response_message`	`"Transaction Approved"` or `None`
`option_1`	`"a1b2c3"`
`option_2`	
`option_3`	
`invoice_number`	`"19832261"`
`invoice_reference`	`"19832261-AA12/1"`
`total_amount`	`"1000"`
`transaction_id`	`"7654321"`
`transaction_status`	`True`
`error_message`	
`token_customer_id`	`"1234567890123456"`
`beagle_score`	`10.23`

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
```

```
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*

- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`

        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's Active-Merchant.

Overview Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

```
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')

- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
```

```
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)
```

```
    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
```

```
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS
```

```
    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
```

```
                            month=10, year=2012,
                            number="4242424242424242",
                            verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant

website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
```

```
...         "item_name_2": "Test Item 2",
...         "amount_2": "20",
...         "invoice": "UID",
...         "notify_url": "http://example.com/paypal-ipn-handler/",
...         "return_url": "http://example.com/paypal/",
...         "cancel_return": "http://example.com/paypal/unsuccessful/",
...     })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
```

```
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
```

```
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
```

```
        braintree_obj.add_fields(fields)
        return render_to_response("some_template.html",
                                  {"bp": braintree_obj},
                                  context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration *Stripe Payment Integration* is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
        from billing.integrations.stripe_integration import StripeIntegration

        class StripeExampleIntegration(StripeIntegration):
            def transaction(self, request):
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
```

```
                ...
            else:
                # Transaction failed
                ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
        int_obj = get_integration("authorize_net_dpm")
        fields = {'x_amount': 1,
                  'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
                  'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
```

```
                'x_recurring_bill': 'F',
               }
    int_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"adp": int_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations

- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors

Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's ActiveMerchant.

Overview

Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant

You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card

The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.

  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways

Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.

- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.

- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing

Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway

This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True         # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "authorize_net": {
            "LOGIN_ID" : "???",
            "TRANSACTION_KEY" : "???"
        }
        ...
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("authorize_net")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream

Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
```

```
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway

The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
```

```
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server

Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True
```

```
    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee

Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
```

```
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway

The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway

Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
```

```
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments

Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
```

```
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

WePay Payments

WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
```

```
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
```

```
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing

Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard

PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay

WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
```

```
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service

Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.

- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
```

```
                                        reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect

Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
    from billing import get_integration

    braintree = get_integration("braintree_payments")

    urlpatterns += patterns('',
        (r'^braintree/', include(braintree.urls)),
    )
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.

- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration

Stripe Payment Integration is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting

- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration

The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in payment:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify ip_address. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple (access_code, customer). access_code is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

customer is a dict containing information about the customer. This is particularly useful if you make use of save_token and token_customer_id to save customer details on eWAY's servers. Keys in the dict are:

- token_customer_id
- save_token
- reference
- title
- first_name
- last_name
- company_name
- job_description

- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
```

```
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method

Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.

- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals

The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:

  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.

- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway

Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
```

```
    ...

    def capture(self, money, authorization, options = None):
    ...

    def void(self, identification, options = None):
    ...

    def credit(self, money, identification, options = None):
    ...

    def recurring(self, money, creditcard, options = None):
    ...

    def store(self, creditcard, options = None):
    ...

    def unstore(self, identification, options = None):
    ...
```

Customizing Merchant

While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
```

```
                                        {"resp": response},
                                        context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant

While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

Merchant: Pluggable and Unified API for Payment Processors

Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's ActiveMerchant.

Overview

Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Installing Merchant

You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration

To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite

By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

Credit Card

The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.

  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the** *first_name* and *last_name*. This is also available as a property.

Subclasses

Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"

  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

Gateways

Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

On-site Processing

Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

Authorize.Net Gateway

This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

Beanstream

Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

Bitcoin Gateway

The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server

Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True

    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

Chargebee

Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
```

```
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

eWay Gateway

The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Paylane Gateway

Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.

- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

Stripe Payments

Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

WePay Payments

WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
```

```
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

Off-site Processing

Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration

An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.

- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard

PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode

By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)
```

```
MERCHANT_SETTINGS = {
    ...,
    'pay_pal': {
        "WPP_USER" : '...',
        "WPP_PASSWORD" : '...',
        "WPP_SIGNATURE" : '...',
        "RECEIVER_EMAIL" : '...',
        # Below attribute is optional
        "ENCRYPTED": True
    }
    ...}

PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example

In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

WorldPay

WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example

In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

Amazon Flexible Payment Service

Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.

- AWS_SECRET_ACCESS_KEY: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the AmazonFpsIntegration class:

- __init__(options = {}): The constructor takes a dictionary of options that are used to initialize the underlying FPSConnection that is bundled with boto.
- service_url: A property that returns the API Endpoint depending on whether the the integration is in test_mode or not.
- link_url: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- purchase(amount, options={}): The method that charges a customer right away for the amount amount after receiving a successful token from Amazon. The options dictionary is generated from the return_url on successful redirect from the Amazon payments page. This method returns a dictionary with two items, status representing the status and response representing the response as described by boto.fps.response.FPSResponse.
- authorize(amount, options={}): Similar to the purchase method except that it reserves the payment and doesn't not charge until a capture (settle) is not called. The response is the same as that of purchase.
- capture(amount, options={}): Captures funds from an authorized transaction. The response is the same as the above two methods.
- credit(amount, options={}): Refunds a part of full amount of the transaction.
- void(identification, options={}): Cancel/Null an authorized transaction.
- fps_ipn_handler: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the AmazonFPSResponse model.
- fps_return_url: This method verifies the source of the return URL from Amazon and directs to the transaction.
- transaction: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example

In any app that is present in the settings.INSTALLED_APPS, subclass the AmazonFpsIntegration and implement the transaction method. The file should be available under <app>/integrations/<integration_name>_integration.py:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
```

```
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect

Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI

compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
```

```
                                        RequestSite(request).domain)
            }
    braintree_obj.add_fields(fields)
    return render_to_response("some_template.html",
                              {"bp": braintree_obj},
                              context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

Stripe Payment Integration

Stripe Payment Integration is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
```

```
                # The token is received in the POST request
                resp = self.gateway.purchase(100, request.POST["stripeToken"])
                if resp["status"] == "SUCCESS":
                    # Redirect if the transaction is successful
                    ...
                else:
                    # Transaction failed
                    ...
```

In the views.py:

```
        stripe_obj = get_integration("stripe_example")
        return render_to_response("some_template.html",
                                  {"stripe_obj": stripe_obj},
                                   context_instance=RequestContext(request))
```

In the urls.py:

```
    stripe_obj = get_integration("stripe_example")
    urlpatterns += patterns('',
       (r'^stripe/', include(stripe_obj.urls)),
    )
```

In the template:

```
    {% load render_integration from billing_tags %}
    {% render_integration stripe_obj %}
```

eWAY Payment Integration

The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)
Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using `check_transaction()`.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
token_customer_id	
save_token	
reference	
title	required for save_token
first_name	required for save_token
last_name	required for save_token
company_name	
job_description	
street	
city	
state	
postal_code	
country	required for save_token
email	
phone	
mobile	
comments	
fax	
url	

Supported keys in `payment`:

Key	Notes
total_amount	*required* (**must be cents**)
invoice_number	
invoice_description	
invoice_reference	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
    def payment(request):
        integration = get_integration("eway_au")
        access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
        # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`
- `last_name`
- `company_name`
- `job_description`
- `street`
- `city`
- `state`
- `postal_code`
- `country` – e.g. `au`
- `email`
- `phone`
- `mobile`
- `comments`
- `fax`
- `url`
- `card_number` – e.g. `444433XXXXXX1111`
- `card_name`
- `card_expiry_month`
- `card_expiry_year`

check_transaction()

Check with eWAY what happened with a transaction.

This method requires `access_code` has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

Authorize.Net Direct Post Method

Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
    from billing import get_integration

    integration = get_integration("authorize_net_dpm")

    urlpatterns += patterns('',
       (r'^authorize_net/', include(integration.urls)),
    )
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

Signals

The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*

- *credit*
- *void*
- *store*
- *unstore*

response is the actual response object that is sent after the success.

Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway

Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant

While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

Contributing to Merchant

While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay

        * PayPal
        * Stripe Payments
        * Paylane
        * WePay
        * Beanstream
        * Chargebee
        * Bitcoin
        * `Global Iris`
    - Off-site Processing
        * PayPal
        * `Google Checkout`
        * RBS WorldPay
        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

1.1.2 Indices and tables

- genindex
- modindex
- search

1.1.3 Merchant: Pluggable and Unified API for Payment Processors

Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's ActiveMerchant.

1.1.4 Overview

Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

1.1.5 Installing Merchant

You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

Configuration

To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

Running the Test Suite

By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

1.1.6 Credit Card

The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.

- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

Subclasses

Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')

- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')
- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

1.1.7 Gateways

Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.

- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.
- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transaction id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.

- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.
- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

1.1.8 On-site Processing

Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

1.1.9 Authorize.Net Gateway

This gateway implements the Authorize.Net Advanced Integration Method (AIM).

Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

1.1.10 Beanstream

Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
```

```
    ...
}
```

Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = beanstream.purchase(1000, credit_card)
>>> resp["response"].resp.approved()
True

# Authorize the card for 1000 USD
>>> resp = beanstream.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = beanstream.capture(900, resp["response"].resp["trnId"])
>>> response["response"].resp.approved()
True

# Void an authorized transaction
>>> beanstream.void(resp["response"].resp["trnId"])
```

1.1.11 Bitcoin Gateway

The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
```

```
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

1.1.12 Braintree Payments Server to Server

Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
```

```
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

1.1.13 Chargebee

Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
```

```
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
>>> response["response"]["subscription"]["status"]
'cancelled'

# Bill the user for 1000 USD
# Technically, Chargebee doesn't have a one shot purchase.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more
>>> resp = chargebee.purchase(1000, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Purchase"})
>>> resp["response"]["invoice"]["subscription_id"]
...

# Authorize the card for 100 USD
# Technically, Chargebee doesn't have a one shot authorize.
# Create a plan (called 'oneshot' below) that does a recurring
# subscription with an interval of a decade or more and authorizes
# the card for a large amount
>>> resp = chargebee.authorize(100, credit_card,
    options = {"plan_id": "oneshot", "description": "Quick Authorize"})

# Capture funds (90 USD) from a previously authorized transaction
>>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
>>> response["status"]
'SUCCESS'

# Void an authorized transaction
>>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
>>> resp["status"]
'SUCCESS'
```

1.1.14 eWay Gateway

The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

1.1.15 Paylane Gateway

Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
```

```
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

Example:

Simple usage:

```
>>> paylane = get_gateway("paylane")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = paylane.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = paylane.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = paylane.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = paylane.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

1.1.16 PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
```

```
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

1.1.17 Stripe Payments

Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
>>> response["status"]
SUCCESS


# Store Customer and Credit Card information in the vault
>>> resp = stripe.store(credit_card)

# Unstore a previously stored credit card from the vault
>>> response = stripe.unstore(resp["response"].id)
>>> response["status"]
SUCCESS

# A recurring plan charge
>>> options = {"plan_id": "gold"}
>>> resp = stripe.recurring(credit_card, options = options)
>>> resp["status"]
SUCCESS
```

1.1.18 WePay Payments

WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments

for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
```

```
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

1.1.19 Off-site Processing

Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

Integration

An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.

- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function

Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

1.1.20 PayPal Website Payments Standard

PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

Test or Live Mode

By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
```

```
    }
    ...}

PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

Example

In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...   "business": "paypalemail@somedomain.com",
...   "item_name": "Test Item",
...   "invoice": "UID",
...   "notify_url": "http://example.com/paypal-ipn-handler/",
...   "return_url": "http://example.com/paypal/",
...   "cancel_return": "http://example.com/paypal/unsuccessful/",
...   "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...       "business": "paypalemail@somedomain.com",
...       "item_name_1": "Test Item 1",
...       "amount_1": "10",
...       "item_name_2": "Test Item 2",
...       "amount_2": "20",
...       "invoice": "UID",
...       "notify_url": "http://example.com/paypal-ipn-handler/",
...       "return_url": "http://example.com/paypal/",
...       "cancel_return": "http://example.com/paypal/unsuccessful/",
...   })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
```

```
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_ca
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

1.1.21 WorldPay

WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

Example

In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
```

```
...                                 {"obj": world_pay},
...                                 context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

1.1.22 Amazon Flexible Payment Service

Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
```

```
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

Example

In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
            return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

1.1.23 Braintree Payments Transparent Redirect

Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

1.1.24 Stripe Payment Integration

Stripe Payment Integration is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

1.1.25 eWAY Payment Integration

The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)
Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)
Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment

- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`

- reference
- title
- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

1.1.26 Authorize.Net Direct Post Method

Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.

- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

1.1.27 Signals

The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

1.1.28 Writing a new gateway

Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

1.1.29 Customizing Merchant

While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

1.1.30 Contributing to Merchant

While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

1.1.31 Changes

0.4 (upcoming)

- Added python3 support
- Removed google checkout

0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

0.1

- Added PIN payments support

0.09

- Removed Samurai gateway and integration

0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

0.07

- Added Chargebee support
- Added Beanstream gateway

0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

0.05

- Added Paylane gateway support.

0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

0.03

- Added support for Stripe and Samurai gateways and integrations.

0.02

- Added a setup.py and uploaded the package to pypi

0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS

        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

1.2 Indices and tables

- genindex
- modindex
- search

1.3 Merchant: Pluggable and Unified API for Payment Processors

Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's ActiveMerchant.

1.4 Overview

Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
```

```
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

1.5 Installing Merchant

You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

1.5.1 Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

1.5.2 Configuration

To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

1.5.3 Running the Test Suite

By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

1.6 Credit Card

The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

1.6.1 Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

1.6.2 Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.
- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

1.6.3 Subclasses

Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*
  - card_name = "Solo"
  - regexp = re.compile('^6767d{12}(d{2,3})?$')
- *Dankort*
  - card_name = "Dankort"
  - regexp = re.compile('^5019d{12}$')
- *Maestro*
  - card_name = "Maestro"
  - regexp = re.compile('^(5[06-8]|6d)d{10,17}$')
- *Forbrugsforeningen*
  - card_name = "Forbrugsforeningen"
  - regexp = re.compile('^600722d{10}$')

- *Laser*
  - card_name = "Laser"
  - regexp = re.compile('^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$')

Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

1.7 Gateways

Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

1.7.1 Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

1.7.2 Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.

- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.

- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

1.7.3 Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

1.8 On-site Processing

Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

1.9 Authorize.Net Gateway

This gateway implements the Authorize.Net Advanced Integration Method (AIM).

1.9.1 Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
```

```
    >>> response1 = g1.purchase(1, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

1.10 Beanstream

Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

1.10.1 Example:

Simple usage:

```
    >>> beanstream = get_gateway("beanstream")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True
```

```
    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

1.11 Bitcoin Gateway

The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

1.11.1 Usage

- Add the following attributes to your *settings.py*:

```
    "bitcoin": {
        "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
        "RPCPASSWORD": "",
        "HOST": "",
        "PORT": "",
        "ACCOUNT": "",
        "MINCONF": 1,
    },
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("bitcoin")
    >>> addr = g1.get_new_address()
    >>> # pass along this address to your customer
    >>> # the purchase will only be successful when
    >>> # the amount is transferred to the above address
    >>> response1 = g1.purchase(100, addr, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

1.12 Braintree Payments Server to Server

Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

1.12.1 Example:

Simple usage:

```
>>> braintree = get_gateway("braintree_payments")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = braintree.purchase(1000, credit_card)
>>> resp["response"].is_success
True

# Authorize the card for 1000 USD
>>> resp = braintree.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = braintree.capture(900, resp["response"].transaction.id)
>>> response["response"].is_success
True

# Void an authorized transaction
>>> braintree.void(resp["response"].transaction.id)

# Store Customer and Credit Card information in the vault
>>> options = {
        "customer": {
            "name": "John Doe",
            "email": "john.doe@example.com",
            },
        }
>>> resp = braintree.store(credit_card, options = options)

# Unstore a previously stored credit card from the vault
>>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
>>> response["response"].is_success
```

```
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

1.13 Chargebee

Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

1.13.1 Example:

Simple usage:

```
    >>> chargebee = get_gateway("chargebee")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")
```

```
    # Bill the user for 10 USD per month based on a plan called 'monthly'
    # The 'recurring' method on the gateway is a mirror to the 'store' method
    >>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
    >>> resp["response"]["customer"]["subscription"]["id"]
    ...

    # Cancel the existing subscription
    >>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

1.14 eWay Gateway

The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

1.14.1 Usage

- Add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True
    MERCHANT_SETTINGS = {
        "eway": {
            "CUSTOMER_ID": "???",
```

```
            "USERNAME": "???",
            "PASSWORD": "???",
        }
    }
```

- Use the gateway instance:

```
    >>> g1 = get_gateway("eway")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {...})
    >>> response1
    {"status": "SUCCESS", "response": <instance>}
```

1.15 Paylane Gateway

Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

1.15.1 Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
```

```
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

1.16 PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

1.16.1 Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
    MERCHANT_TEST_MODE = True    # Toggle for live transactions
    MERCHANT_SETTINGS = {
        "pay_pal": {
            "WPP_USER" : "???",
            "WPP_PASSWORD" : "???",
            "WPP_SIGNATURE" : "???"
        }
    }

    # Since merchant relies on django-paypal
    # you have to additionally provide the
    # below attributes
    PAYPAL_TEST = MERCHANT_TEST_MODE
    PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
    PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
    PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
    >>> g1 = get_gateway("pay_pal")
    >>>
    >>> cc = CreditCard(first_name= "Test",
    ...                 last_name = "User",
    ...                 month=10, year=2011,
    ...                 number="4222222222222",
    ...                 verification_value="100")
    >>>
    >>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
    >>> response1
    {"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

1.17 Stripe Payments

Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

1.17.1 Example:

Simple usage:

```
    >>> from billing import get_gateway, CreditCard
    >>> stripe = get_gateway("stripe")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = stripe.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS

    # Authorize the card for 1000 USD
```

```
    >>> resp = stripe.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = stripe.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

1.18 WePay Payments

WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
```

```
    ...
}
```

1.18.1 Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
```

```
                "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

1.19 Off-site Processing

Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

1.19.1 Integration

An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.
- **urls**: A property that returns the above method.

Helper Function

Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

1.20 PayPal Website Payments Standard

PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

1.20.1 Test or Live Mode

By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

1.20.2 Example

In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
```

```
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

1.21 WorldPay

WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

1.21.1 Example

In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

1.22 Amazon Flexible Payment Service

Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.

- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.
- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

1.22.1 Example

In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
```

```
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
   if request.is_secure():
       url_scheme = "https"
   domain = RequestSite(request).domain
   fields = {"transactionAmount": "100",
             "pipelineName": "SingleUse",
             "paymentReason": "Merchant Test",
             "paymentPage": request.build_absolute_uri(),
             # Send the correct url where the redirect should happen
             "returnURL": "%s://%s%s" % (url_scheme,
                                         domain,
                                         reverse("fps_return_url")),
            }
    # You might want to save the fields["callerReference"] that
    # is auto-generated in the db or session to uniquely identify
    # this user (or use the user id as the callerReference) because
    # amazon passes this callerReference back in the return URL.
    amazon_fps.add_fields(fields)
    return render_to_response("some_template.html",
                              {"fps": amazon_fps},
                              context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

1.23 Braintree Payments Transparent Redirect

Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.

- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields` (added through either the `add_fields` or `add_field` methods) and `tr_data`.

1.23.1 Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

1.24 Stripe Payment Integration

Stripe Payment Integration is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

1.24.1 Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
            # Transaction failed
            ...
```

In the views.py:

```
stripe_obj = get_integration("stripe_example")
return render_to_response("some_template.html",
                          {"stripe_obj": stripe_obj},
                           context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

1.25 eWAY Payment Integration

The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country
- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`
- `first_name`

- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

1.25.1 Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

1.26 Authorize.Net Direct Post Method

Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.

- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

1.26.1 Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

1.27 Signals

The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

1.28 Writing a new gateway

Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

1.29 Customizing Merchant

While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

and the Integration class name in that file as `ModifiedBpIntegration`.

Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
        return render_to_response("my_new_success.html",
                                  {"resp": response},
                                  context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

1.30 Contributing to Merchant

While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

1.31 Changes

1.31.1 0.4 (upcoming)

- Added python3 support
- Removed google checkout

1.31.2 0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

1.31.3 0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

1.31.4 0.1

- Added PIN payments support

1.31.5 0.09

- Removed Samurai gateway and integration

1.31.6 0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net
- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

1.31.7 0.07

- Added Chargebee support
- Added Beanstream gateway

1.31.8 0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

1.31.9 0.05

- Added Paylane gateway support.

1.31.10 0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

1.31.11 0.03

- Added support for Stripe and Samurai gateways and integrations.

1.31.12 0.02

- Added a setup.py and uploaded the package to pypi

1.31.13 0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay

        * Amazon FPS
        * Braintree Payments Transparent Redirect
        * Stripe
        * eWAY
        * Authorize.Net Direct Post Method
        * `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Merchant: Pluggable and Unified API for Payment Processors

Merchant, is a django app that offers a uniform api and pluggable interface to interact with a variety of payment processors. It is heavily inspired from Ruby's ActiveMerchant.

Overview

Simple how to:

```
# settings.py
# Authorize.Net settings
AUTHORIZE_LOGIN_ID = "..."
AUTHORIZE_TRANSACTION_KEY = "..."

# PayPal settings
PAYPAL_TEST = True
PAYPAL_WPP_USER = "..."
PAYPAL_WPP_PASSWORD = "..."
PAYPAL_WPP_SIGNATURE = "..."

# views.py or wherever you want to use it
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User,
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
>>>
>>> g2 = get_gateway("pay_pal")
>>>
>>> response2 = g2.purchase(100, cc, options = {...})
>>> response2
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

CHAPTER 5

Installing Merchant

You can use any of the following methods to install merchant.

- The recommended way is to install from PyPi:

```
pip install django-merchant
```

- If you are feeling adventurous, you might want to run the code off the git repository:

```
pip install -e git+git://github.com/agiliq/merchant.git#egg=django-merchant
```

5.1 Post-installation

- Install the dependencies for the gateways as prescribed in the individual gateway doc.
- Reference the `billing` app in your settings `INSTALLED_APPS`.
- Run `python manage.py syncdb` to create the new required database tables

5.2 Configuration

To configure a gateway/integration add the corresponding key to `MERCHANT_SETTINGS`. Take a look at `local.py-dist` for reference.

5.3 Running the Test Suite

By default, the test suite is configured to run tests for all the gateways and integrations which are configured:

```
python manage.py test billing
```

Tests for gateways and integrations which are not configured will be skipped.

If you are planning to integrate your app with a specific gateway/integration then you might wish to run only that apps test suite. For example, to run the `Google Checkout Integration` test case:

```
python manage.py test billing.GoogleCheckoutTestCase
```

CHAPTER 6

Credit Card

The *CreditCard* class is a helper class with some useful methods mainly for validation. This class is available in *billing.utils.credit_card*.

6.1 Attribute Reference

- *regexp*: The compiled regular expression that matches all card numbers for the card issuing authority. For the *CreditCard* class, this is *None*. It is overridden by subclasses.
- *card_type*: Points to a one of *CreditCard*'s subclasses. This attribute is set by the *validate_card* method of the selected gateway.
- *card_name*: Card issuing authority name. Generally not required, but some gateways expect the user to figure out the credit card type to send with the requests.

6.2 Method Reference

- *__init__*: This method expects 6 keyword arguments. They are
  - *first_name*: The first name of the credit card holder.
  - *last_name*: The last name of the credit card holder.
  - *cardholders_name*: The full name of the credit card holder, as an alternative to supplying *first_name* and *last_name*.
  - *month*: The expiration month of the credit card as an integer. **Required**
  - *year*: The expiration year of the credit card as an integer. **Required**
  - *number*: The credit card number (generally 16 digits). **Required**
  - *verification_value*: The card security code (CVV2). **Required**
- *is_luhn_valid*: Checks the validity of the credit card number by using the *Luhn's algorithm* and returns a boolean. This method takes no arguments.
- *is_expired*: Checks if the expiration date of the card is beyond today and returns a boolean. This method takes no arguments.
- *valid_essential_attributes*: Verifies if all the 6 arguments provided to the *__init__* method are filled and returns a boolean.

- *is_valid*: Checks the validity of the card by calling the *is_luhn_valid*, *is_expired* and *valid_essential_attributes* method and returns a boolean. This method takes no arguments.
- *expire_date*: Returns the card expiry date in the "MM-YYYY" format. This is also available as a property.
- ***name*: Returns the full name of the credit card holder by concatenating the *first_name* and *last_name*.** This is also available as a property.

6.3 Subclasses

Normally you do not use the subclasses directly. Instead, you use *CreditCard*, and call gateway.validate_card() which will add a *card_type* attribute which is the subclass.

The various credit cards and debit cards supported by Merchant are:

6.3.1 Credit Cards

- *Visa*
  - card_name = "Visa"
  - regexp = re.compile('^4d{12}(d{3})?$')
- *MasterCard*
  - card_name = "MasterCard"
  - regexp = re.compile('^(5[1-5]d{4}|677189)d{10}$')
- *Discover*
  - card_name = "Discover"
  - regexp = re.compile('^(6011|65d{2})d{12}$')
- *AmericanExpress*
  - card_name = "Amex"
  - regexp = re.compile('^3[47]d{13}$')
- *DinersClub*
  - card_name = "DinersClub"
  - regexp = re.compile('^3(0[0-5]|[68]d)d{11}$')
- *JCB*
  - card_name = "JCB"
  - regexp = re.compile('^35(28|29|[3-8]d)d{12}$')

6.3.2 Debit Cards

- *Switch*
  - card_name = "Switch"
  - regexp = re.compile('^6759d{12}(d{2,3})?$')
- *Solo*

  - card_name = “Solo”
  - regexp = re.compile(‘^6767d{12}(d{2,3})?$’)
- *Dankort*
  - card_name = “Dankort”
  - regexp = re.compile(‘^5019d{12}$’)
- *Maestro*
  - card_name = “Maestro”
  - regexp = re.compile(‘^(5[06-8]|6d)d{10,17}$’)
- *Forbrugsforeningen*
  - card_name = “Forbrugsforeningen”
  - regexp = re.compile(‘^600722d{10}$’)
- *Laser*
  - card_name = “Laser”
  - regexp = re.compile(‘^(6304|6706|6771|6709)d{8}(d{4}|d{6,7})?$’)

6.3.3 Helpers

- all_credit_cards = [Visa, MasterCard, Discover, AmericanExpress, DinersClub, JCB]
- all_debit_cards = [Switch, Solo, Dankort, Maestro, Forbrugsforeningen, Laser]
- all_cards = all_credit_cards + all_debit_cards

CHAPTER 7

Gateways

Gateways are the payment processors implemented in Merchant. This is implemented as a class so that it is easy to extend and create as many gateways as possible.

The base gateway class is *billing.gateway.Gateway* which has the following methods and attributes.

7.1 Attribute Reference

- **test_mode**: This boolean attribute signifies if the gateway is in the test mode. By default, it looks up this value from the *MERCHANT_TEST_MODE* attribute from the settings file. If the *MERCHANT_TEST_MODE* attribute is not found in the settings file, the default value is *True* indicating that the gateway is in the test mode. So do not forget to either set the attribute to *True* in the subclass or through the settings file.
- **default_currency**: This is the currency in which the transactions are settled ie the currency in which the payment gateway sends the invoice, transaction reports etc. This does not prevent the developer from charging a customer in other currencies but the exchange rate conversion has to be manually handled by the developer. This is a string, for example *"USD"* for US Dollar.
- **supported_countries**: This is a *list* of supported countries that are handled by the payment gateway. This should contain a list of the country codes as prescribed by the ISO 3166-alpha 2 standard. The *billing.utils.countries* contains a mapping of the country names and ISO codes.
- **supported_cardtypes**: This is a *list* of supported card types handled by the payment gateway. This should contain a list of instances of the CreditCard class.
- **homepage_url**: A string pointing to the URL of the payment gateway. This is just a helper attribute that is currently not used.
- **display_name**: A string that contains the name of the payment gateway. Another helper attribute that is currently not used.
- **application_id**: An application name or unique identifier for the gateway. Yet another helper attribute not currently used.

7.2 Method Reference

- **validate_card(credit_card)**: This method validates the supplied card by checking if it is supported by the gateway (through the *supported_cardtypes* attribute) and calls the *is_valid* method of the card and returns a boolean. if the card is not supported by the gateway, a *CardNotSupported* exception is raised.

- **service_url**: A property that returns the url to which the credit card and other transaction related details are submitted.
- **purchase(money, credit_card, options = None)**: A method that charges the given card (one-time) for the given amount *money* using the *options* provided. Subclasses have to implement this method.
- **authorize(money, credit_card, options = None)**: A method that authorizes (for a future transaction) the credit card for the amount *money* using the *options* provided. Subclasses have to implement this method.
- **capture(money, authorization, options = None)**: A method that captures funds from a previously authorized transaction using the *options* provided. Subclasses have to implement this method.
- **void(identification, options = None)**: A method that nulls/voids/blanks an authorized transaction identified by *identification* to prevent a subsequent capture. Subclasses have to implement this method.
- **credit(money, identification, options = None)**: A method that refunds a settled transaction with the transacation id *identification* and given *options*. Subclasses must implement this method.
- **recurring(money, creditcard, options = None)**: A method that sets up a recurring transaction (or a subscription). Subclasses must implement this method.
- **store(creditcard, options = None)**: A method that stores the credit card and user profile information on the payment gateway's servers for future reference. Subclasses must implement this method.
- **unstore(identification, options = None)**: A method that reverses the *store* method's results. Subclasses must implement this method.

The *options* dictionary passed to the above methods consists of the following keys:

- **order_id**: A unique order identification code (usually set by the gateway).
- **ip**: The IP address of the customer making the purchase. This is required by certain gateways like PayPal.
- **customer**: The name, customer number, or other information that identifies the customer. Optional.
- **invoice**: The invoice code/number (set by the merchant).
- **merchant**: The name or description of the merchant offering the product.
- **description**: A description of the product or transaction.
- **email**: The email address of the customer. Required by a few gateways.
- **currency**: Required when using a currency with a gateway that supports multiple currencies. If not specified, the value of the *default_currency* attribute of the gateway instance is used.
- **billing_address**: A dictionary containing the billing address of the customer. Generally required by gateways for address verification (AVS) etc.
- **shipping_address**: A dictionary containing the shipping address of the customer. Required if the merchant requires shipping of products and where billing address is not the same as shipping address.

The address dictionary for *billing_address* and *shipping_address* should have the following keys:

- **name**: The full name of the customer.
- **company**: The company name of the customer. Required by a few gateways.
- **address1**: The primary street address of the customer. Required by many gateways.
- **address2**: Additional line for the address. Optional.
- **city**: The city of the customer.
- **state**: The state of the customer.
- **country**: The ISO 3166-alpha 2 standard code for the country of the customer.

- **zip**: The zip or postal code of the customer.
- **phone**: The phone number of the customer. Optional.

All the above methods return a standard *response* dictionary containing the following keys:

- **status**: Indicating if the transaction is a "**SUCCESS**" or a "**FAILURE**"
- **response**: The response object for the transaction. Please consult the respective gateway's documentation to learn more about it.

7.3 Helper functions

- **get_gateway(name, *args, **kwargs)**: A helper function that loads the gateway class by the *name* and initializes it with the *args* and *kwargs*.

CHAPTER 8

On-site Processing

Onsite processing refers to the payment mechanism where the customer stays on the merchant website and the authentication is done by the merchant website with the gateway in the background.

Merchant websites need to comply with PCI standards to be able to securely carry out transactions.

On-site processing for payment gateways is implemented by using subclasses of the Gateway class.

CHAPTER 9

Authorize.Net Gateway

This gateway implements the Authorize.Net Advanced Integration Method (AIM).

9.1 Usage

- Setup a test account with Authorize.Net.
- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True         # Toggle for live transactions
MERCHANT_SETTINGS = {
    "authorize_net": {
        "LOGIN_ID" : "???",
        "TRANSACTION_KEY" : "???"
    }
    ...
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("authorize_net")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(1, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <AuthorizeNetAIMResponse object>}
```

CHAPTER 10

Beanstream

Beanstream is a gateway headquartered in Canada and offering payment processing across North America.

Note: You will require the beanstream python package maintained by the community.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `MERCHANT_ID`: The merchant id provided by Beanstream. Can be obtained from the account dashboard.
- `LOGIN_COMPANY`: The company name as visible from the account settings in the dashboard.
- `LOGIN_USER`: The username used to login to the account dashboard.
- `LOGIN_PASSWORD`: The password used to login to the account dashboard.
- `HASH_ALGORITHM`: This is optional but required if you have enabled hashing in account dashboard. The values may be one of *SHA-1* and *MD5*.
- `HASHCODE`: If the above attribute is enabled, then set this attribute to the hash value you've setup in the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "beanstream": {
        "MERCHANT_ID": "???",
        "LOGIN_COMPANY": "???",
        "LOGIN_USER": "???",
        "LOGIN_PASSWORD": "???",
        # The below two attributes are optional
        "HASH_ALGORITHM": "???",
        "HASHCODE": "???",
    }
    ...
}
```

10.1 Example:

Simple usage:

```
>>> beanstream = get_gateway("beanstream")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
```

```
                                number="4111111111111111",
                                verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = beanstream.purchase(1000, credit_card)
    >>> resp["response"].resp.approved()
    True

    # Authorize the card for 1000 USD
    >>> resp = beanstream.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = beanstream.capture(900, resp["response"].resp["trnId"])
    >>> response["response"].resp.approved()
    True

    # Void an authorized transaction
    >>> beanstream.void(resp["response"].resp["trnId"])
```

CHAPTER 11

Bitcoin Gateway

The Bitcoin gateway implements the Bitcoin digital currency.

It is implemented using the JSON-RPC API as described in the Merchant Howto.

Note: The Bitcoin gateway depends on the *bitcoin-python* library which can be installed from pypi

11.1 Usage

- Add the following attributes to your *settings.py*:

```
"bitcoin": {
    "RPCUSER": "", # you'll find these settings in your $HOME/.bitcoin/bitcoin.conf
    "RPCPASSWORD": "",
    "HOST": "",
    "PORT": "",
    "ACCOUNT": "",
    "MINCONF": 1,
},
```

- Use the gateway instance:

```
>>> g1 = get_gateway("bitcoin")
>>> addr = g1.get_new_address()
>>> # pass along this address to your customer
>>> # the purchase will only be successful when
>>> # the amount is transferred to the above address
>>> response1 = g1.purchase(100, addr, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

Braintree Payments Server to Server

Braintree Payments Server to Server is a gateway provided by Braintree Payments to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official braintree python package offered by Braintree for this gateway to work.

Settings attributes required for this integration are:

- `MERCHANT_ACCOUNT_ID`: The merchant account id provided by Braintree. Can be obtained from the account dashboard.
- `PUBLIC_KEY`: The public key provided by Braintree through their account dashboard.
- `PRIVATE_KEY`: The private key provided by Braintree through their account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "braintree_payments": {
        "MERCHANT_ACCOUNT_ID": "???",
        "PUBLIC_KEY": "???",
        "PRIVATE_KEY": "???"
    }
    ...
}
```

12.1 Example:

Simple usage:

```
    >>> braintree = get_gateway("braintree_payments")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2011,
                                 number="4111111111111111",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = braintree.purchase(1000, credit_card)
    >>> resp["response"].is_success
    True
```

```
    # Authorize the card for 1000 USD
    >>> resp = braintree.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = braintree.capture(900, resp["response"].transaction.id)
    >>> response["response"].is_success
    True

    # Void an authorized transaction
    >>> braintree.void(resp["response"].transaction.id)

    # Store Customer and Credit Card information in the vault
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            }
    >>> resp = braintree.store(credit_card, options = options)

    # Unstore a previously stored credit card from the vault
    >>> response = braintree.unstore(resp["response"].customer.credit_cards[0].token)
    >>> response["response"].is_success
    True

    # A recurring plan charge
    >>> options = {
            "customer": {
                "name": "John Doe",
                "email": "john.doe@example.com",
                },
            "recurring": {
                "plan_id": "test_plan",
                "trial_duration": 2,
                "trial_duration_unit": "month",
                "number_of_billing_cycles": 12,
                },
            }
    >>> resp = braintree.recurring(10, credit_card, options = options)
    >>> resp["response"].is_success
    True
    >>> resp["response"].subscription.number_of_billing_cycles
    12
```

CHAPTER 13

Chargebee

Chargebee is a SAAS that makes subscription billing easy to handle. They also provide the functionality to plug to multiple gateways in the backend.

Note: You will require the requests package to get Chargebee to work.

Settings attributes required (optional if you are passing them while initializing the gateway) for this integration are:

- `SITE`: The name of the Chargebee app (or site as they refer). The URL is generally of the form "https://{site}.chargebee.com/".
- `API_KEY`: This key is provided in your settings dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "chargebee": {
        "SITE": "some-test",
        "API_KEY": "???",
    }
    ...
}
```

13.1 Example:

Simple usage:

```
>>> chargebee = get_gateway("chargebee")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2011,
                             number="4111111111111111",
                             verification_value="100")

# Bill the user for 10 USD per month based on a plan called 'monthly'
# The 'recurring' method on the gateway is a mirror to the 'store' method
>>> resp = chargebee.store(credit_card, options = {"plan_id": "monthly"})
>>> resp["response"]["customer"]["subscription"]["id"]
...

# Cancel the existing subscription
>>> response = chargebee.unstore(resp["response"]["customer"]["subscription"]["id"])
```

```
    >>> response["response"]["subscription"]["status"]
    'cancelled'

    # Bill the user for 1000 USD
    # Technically, Chargebee doesn't have a one shot purchase.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more
    >>> resp = chargebee.purchase(1000, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Purchase"})
    >>> resp["response"]["invoice"]["subscription_id"]
    ...

    # Authorize the card for 100 USD
    # Technically, Chargebee doesn't have a one shot authorize.
    # Create a plan (called 'oneshot' below) that does a recurring
    # subscription with an interval of a decade or more and authorizes
    # the card for a large amount
    >>> resp = chargebee.authorize(100, credit_card,
        options = {"plan_id": "oneshot", "description": "Quick Authorize"})

    # Capture funds (90 USD) from a previously authorized transaction
    >>> response = chargebee.capture(90, resp["response"]["subscription"]["id"])
    >>> response["status"]
    'SUCCESS'

    # Void an authorized transaction
    >>> resp = chargebee.void(resp["response"]["invoice"]["subscription_id"])
    >>> resp["status"]
    'SUCCESS'
```

CHAPTER 14

eWay Gateway

The eWay gateway implements the eWay Hosted Payment API.

Note: Since the eWay payment gateway uses SOAP, the API has been implemented using the suds SOAP library for python. You'll require it to be able to use this gateway.

14.1 Usage

- Add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

- Use the gateway instance:

```
>>> g1 = get_gateway("eway")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
>>> response1 = g1.purchase(100, cc, options = {...})
>>> response1
{"status": "SUCCESS", "response": <instance>}
```

CHAPTER 15

Paylane Gateway

Paylane is a payment processor focussed mainly in Europe.

Note: You will require suds python package to work with the the SOAP interface.

Settings attributes required for this gateway are:

- `USERNAME`: The username provided by Paylane while signing up for an account.
- `PASSWORD`: The password you set from the merchant admin panel. Not to be confused with the merchant login password.
- `WSDL` (optional): The location of the WSDL file. Defaults to https://direct.paylane.com/wsdl/production/Direct.wsdl.
- `SUDS_CACHE_DIR` (optional): The location of the suds cache files. Defaults to `/tmp/suds`.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "paylane": {
        "USERNAME": "???",
        "PASSWORD": "???",
    }
    ...
}
```

15.1 Example:

Simple usage:

```
    >>> paylane = get_gateway("paylane")
    >>> credit_card = CreditCard(first_name="Test", last_name="User",
                                 month=10, year=2012,
                                 number="4242424242424242",
                                 verification_value="100")

    # Bill the user for 1000 USD
    >>> resp = paylane.purchase(1000, credit_card)
    >>> resp["status"]
    SUCCESS
```

```
    # Authorize the card for 1000 USD
    >>> resp = paylane.authorize(1000, credit_card)

    # Capture funds (900 USD) from a previously authorized transaction
    >>> response = paylane.capture(900, resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = paylane.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

CHAPTER 16

PayPal Gateway

Note: This gateway is a wrapper to the django-paypal package. Please download it to be able to use the gateway.

The PayPal gateway is an implementation of the PayPal Website Payments Pro product.

16.1 Usage

- Setup a PayPal Website Payments Pro account and obtain the API details.
- Add *paypal.standard* and *paypal.pro* (apps from django-paypal) to the *INSTALLED_APPS* in your *settings.py*.
- Also add the following attributes to your *settings.py*:

```
MERCHANT_TEST_MODE = True    # Toggle for live transactions
MERCHANT_SETTINGS = {
    "pay_pal": {
        "WPP_USER" : "???",
        "WPP_PASSWORD" : "???",
        "WPP_SIGNATURE" : "???"
    }
}

# Since merchant relies on django-paypal
# you have to additionally provide the
# below attributes
PAYPAL_TEST = MERCHANT_TEST_MODE
PAYPAL_WPP_USER = MERCHANT_SETTINGS["pay_pal"]["WPP_USER"]
PAYPAL_WPP_PASSWORD = MERCHANT_SETTINGS["pay_pal"]["WPP_PASSWORD"]
PAYPAL_WPP_SIGNATURE = MERCHANT_SETTINGS["pay_pal"]["WPP_SIGNATURE"]
```

- Run *python manage.py syncdb* to get the response tables.
- Use the gateway instance:

```
>>> g1 = get_gateway("pay_pal")
>>>
>>> cc = CreditCard(first_name= "Test",
...                 last_name = "User",
...                 month=10, year=2011,
...                 number="4222222222222",
...                 verification_value="100")
>>>
```

```
>>> response1 = g1.purchase(100, cc, options = {"request": request, ...})
>>> response1
{"status": "SUCCESS", "response": <PayPalNVP object>}
```

Note: The PayPal gateway expects you pass the *request* object as a part of the *options* dictionary because the client's IP address may be used for fraud detection.

CHAPTER 17

Stripe Payments

Stripe Payments is a gateway provided by Stripe to services which are willing to take the burden of PCI compliance. This does not involve any redirects and only Server to Server calls happen in the background.

Note: You will require the official stripe python package offered by Stripe for this gateway to work.

Settings attributes required for this integration are:

- `API_KEY`: The merchant api key is provided by Stripe. Can be obtained from the account dashboard.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "stripe": {
        "API_KEY": "???",
        "PUBLISHABLE_KEY": "???", # Used for stripe integration
    }
    ...
}
```

17.1 Example:

Simple usage:

```
>>> from billing import get_gateway, CreditCard
>>> stripe = get_gateway("stripe")
>>> credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

# Bill the user for 1000 USD
>>> resp = stripe.purchase(1000, credit_card)
>>> resp["status"]
SUCCESS

# Authorize the card for 1000 USD
>>> resp = stripe.authorize(1000, credit_card)

# Capture funds (900 USD) from a previously authorized transaction
>>> response = stripe.capture(900, resp["response"].id)
```

```
    >>> response["status"]
    SUCCESS


    # Store Customer and Credit Card information in the vault
    >>> resp = stripe.store(credit_card)

    # Unstore a previously stored credit card from the vault
    >>> response = stripe.unstore(resp["response"].id)
    >>> response["status"]
    SUCCESS

    # A recurring plan charge
    >>> options = {"plan_id": "gold"}
    >>> resp = stripe.recurring(credit_card, options = options)
    >>> resp["status"]
    SUCCESS
```

CHAPTER 18

WePay Payments

WePay.com is a service that lets you accept payments not just from credit cards but also from bank accounts.

WePay works slightly differently and is a hybrid between a Gateway and an Integration but should still be fairly easy to use.

Note: You will require the official wepay python package offered by WePay.

Settings attributes required for this integration are:

- `CLIENT_ID`: This attribute refers to the application id that can be obtained from the account dashboard.
- `CLIENT_SECRET`: This is the secret for the corresponding `CLIENT_ID`.
- `ACCOUNT_ID`: Refers to the WePay user account id. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `account_id`) in the views.
- `ACCESS_TOKEN`: The OAuth2 access token acquired from the user after the installation of the WePay application. If you are accepting payments for yourself, then this attribute is compulsory. If you are accepting payments for other users (say in a marketplace setup), then it is optional in the `settings.py` file but has to be passed in the options dictionary (with the key `token`) in the views.

Settings attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "we_pay": {
        "CLIENT_ID": "???",
        "CLIENT_SECRET": "???",
        "ACCESS_TOKEN": "???",
        "ACCOUNT_ID": "???"
    }
    ...
}
```

18.1 Example:

Simple usage:

```
    wp = get_gateway("we_pay")
    credit_card = CreditCard(first_name="Test", last_name="User",
                             month=10, year=2012,
                             number="4242424242424242",
                             verification_value="100")

    def we_pay_purchase(request):
        # Bill the user for 10 USD
        # Credit card is not required here because the user
        # is redirected to the wepay site for authorization
        resp = wp.purchase(10, None, {
            "description": "Product Description",
            "type": "GOODS",
            "redirect_uri": "http://example.com/success/redirect/"
        })
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["checkout_uri"])
        ...

    # Authorize the card for 1000 USD
    def we_pay_authorize(request):
        # Authorize the card, the amount is not required.
        resp = wp.authorize(None, credit_card, {"customer": {"email": "abc@example.com"}, "billing_a
        resp["checkout_id"]
        ...

    # Capture funds from a previously authorized transaction
    def we_pay_capture(request):
        # No ability to partially capture and hence first argument is None
        resp = wp.capture(None, '<authorization_id>')
        ...

    # Refund a transaction
    def we_pay_refund(request):
        # Refund completely
        resp = wp.credit(None, '<checkout_id>')
        ...
        # Refund partially from a transaction charged $15
        resp = wp.credit(10, '<checkout_id>')
        ...

    # Store Customer and Credit Card information in the vault
    def we_pay_store(request)
        resp = wp.store(credit_card, {"customer": {"email": "abc@example.com"}, "billing_address": {
        ...

    # A recurring plan for $100/month
    def we_pay_recurring(request):
        options = {"period": "monthly", "start_time": "2012-01-01",
                   "end_time": "2013-01-01", "auto_recur": "true",
                   "redirect_uri": "http://example.com/redirect/success/"}
        resp = wp.recurring(100, None, options = options)
        if resp["status"] == "SUCCESS":
            return HttpResponseRedirect(resp["response"]["preapproval_uri"])
        ...
```

CHAPTER 19

Off-site Processing

Off-site processing is the payment mechanism where the customer is redirected to the payment gateways site to complete the transaction and is redirected back to the merchant website on completion.

Since the credit card number and other sensitive details are entered on the payment gateway's site, the merchant website may not comply to PCI standards. This mode of payment is recommended when the merchant website is not in a position to use SSL certificates, not able to guarantee a secure network etc

Off-site processing is generally implemented in merchant through *Integrations* (name derived from Active Merchant).

19.1 Integration

An Integration much like a Gateway is a Python class. But unlike a Gateway which is used in a view, an Integration renders a form (usually with hidden fields) through a template tag. An integration may also support asynchronous and real-time transaction status handling through callbacks or notifiers like the PayPal IPN

Here is a reference of the attributes and methods of the Integration class:

19.1.1 Attributes

- **fields**: Dictionary of form fields that have to be rendered in the template.
- **test_mode**: Signifies if the integration is in a test mode or production. The default value for this is taken from the *MERCHANT_TEST_MODE* setting attribute.
- **display_name**: A human readable name that is generally used to tag the errors when the integration is not correctly configured.

19.1.2 Methods

- **__init__(options={})**: The constructor for the Integration. The options dictionary if present overrides the default items of the fields attribute.
- **add_field(key, value)**: A method to modify the fields attribute.
- **add_fields(fields)**: A method to update the fields attribute with the fields dictionary specified.
- **service_url**: The URL on the form where the fields data is posted. Overridden by implementations.
- **get_urls**: A method that returns the urlpatterns for the notifier/ callback. This method is modified by implementations.

- **urls**: A property that returns the above method.

19.1.3 Helper Function

Very much like Gateways, Integrations have a method of easily referencing the corresponding integration class through the *get_integration* helper function.

- **get_integration(integration_name, *args, **kwargs)**: Returns the Integration class for the corresponding *integration_name*.

Example:

```
>>> from billing import get_integration
>>> get_integration("pay_pal")
<billing.integrations.pay_pal_integration.PayPalIntegration object at 0xa57e12c>
```

PayPal Website Payments Standard

PayPal Website Payments Standard (PWS) is an offsite payment processor. This method of payment is implemented in merchant as a wrapper on top of django-paypal. You need to install the package to be able to use this payment processor.

For a list of the fields and settings attribute expected, please refer to the PWS and django-paypal documentation.

After a transaction, PayPal pings the notification URL and all the data sent is stored in the *PayPalIPN* model instance that can be viewed from the django admin.

20.1 Test or Live Mode

By default the form renders in test mode with POST against `sandbox.paypal.com`. Add following to you *settings.py* to put the form into live mode:

```
### Django Merchant
MERCHANT_TEST_MODE = False
PAYPAL_TEST = MERCHANT_TEST_MODE
```

Don't forget to add the settings attributes from `django-paypal`:

```
INSTALLED_APPS = (
    ...,
    'paypal.standard.pdt',
    ...)

 MERCHANT_SETTINGS = {
     ...,
     'pay_pal': {
         "WPP_USER" : '...',
         "WPP_PASSWORD" : '...',
         "WPP_SIGNATURE" : '...',
         "RECEIVER_EMAIL" : '...',
         # Below attribute is optional
         "ENCRYPTED": True
     }
     ...}

 PAYPAL_RECEIVER_EMAIL = MERCHANT_SETTINGS['pay_pal']['RECEIVER_EMAIL']
```

20.2 Example

In urls.py:

```
from billing import get_integration
pay_pal = get_integration("pay_pal")
urlpatterns += patterns('',
  (r'^paypal-ipn-handler/', include(pay_pal.urls)),
)
```

In views.py:

```
>>> from billing import get_integration
>>> pay_pal = get_integration("pay_pal")
>>> pay_pal.add_fields({
...    "business": "paypalemail@somedomain.com",
...    "item_name": "Test Item",
...    "invoice": "UID",
...    "notify_url": "http://example.com/paypal-ipn-handler/",
...    "return_url": "http://example.com/paypal/",
...    "cancel_return": "http://example.com/paypal/unsuccessful/",
...    "amount": 100})
>>> return render_to_response("some_template.html",
...                           {"obj": pay_pal},
...                           context_instance=RequestContext(request))
```

You can also implement a shopping cart by adding multiple items with keys like `item_name_1`, `amount_1` etc, for e.g:

```
>>> pay_pal.add_fields({
...        "business": "paypalemail@somedomain.com",
...        "item_name_1": "Test Item 1",
...        "amount_1": "10",
...        "item_name_2": "Test Item 2",
...        "amount_2": "20",
...        "invoice": "UID",
...        "notify_url": "http://example.com/paypal-ipn-handler/",
...        "return_url": "http://example.com/paypal/",
...        "cancel_return": "http://example.com/paypal/unsuccessful/",
...    })
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form action="https://www.sandbox.paypal.com/cgi-bin/webscr" method="post">
  <input type="hidden" name="business" value="paypalemail@somedomain.com" id="id_business" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="item_name" value="Test Item" id="id_item_name" />
  <input type="hidden" name="notify_url" value="http://example.com/paypal-ipn-handler/" id="id_notify
  <input type="hidden" name="cancel_return" value="http://example.com/paypal/unsuccessful" id="id_can
  <input type="hidden" name="return" value="http://example.com/paypal/" id="id_return_url" />
  <input type="hidden" name="invoice" value="UID" id="id_invoice" />
  <input type="hidden" name="cmd" value="_xclick" id="id_cmd" />
  <input type="hidden" name="charset" value="utf-8" id="id_charset" />
  <input type="hidden" name="currency_code" value="USD" id="id_currency_code" />
```

```
  <input type="hidden" name="no_shipping" value="1" id="id_no_shipping" />
  <input type="image" src="https://www.sandbox.paypal.com/en_US/i/btn/btn_buynowCC_LG.gif" border="0"
</form>
```

CHAPTER 21

WorldPay

WorldPay, provides a hosted payments page for offsite transactions for merchants who cannot guarantee PCI compliance. The documentation for the service is available here.

After a transaction, WorldPay pings the notification URL and all the data sent is stored in the *RBSResponse* model instance that can be viewed from the django admin.

The settings attribute required for this integration are:

- **MD5_SECRET_KEY**: The MD5 secret key chosen by the user while signing up for the WorldPay Hosted Payments Service.

Settings Attributes:

```
MERCHANT_TEST_MODE = True # Toggle for live
MERCHANT_SETTINGS = {
    "world_pay": {
        "MD5_SECRET_KEY": "???"
    }
    ...
}
```

21.1 Example

In urls.py:

```
world_pay = get_integration("world_pay")
urlpatterns += patterns('',
  (r'^world_pay/', include(world_pay.urls)),
  # You'll have to register /world_pay/rbs-notify-handler/ in the
  # WorldPay admin dashboard for the notification URL
)
```

In views.py:

```
>>> from billing import get_integration
>>> world_pay = get_integration("world_pay")
>>> world_pay.add_fields({
...     "instId": "WP_ID",
...     "cartId": "TEST123",
...     "amount": 100,
...     "currency": "USD",
...     "desc": "Test Item",
```

```
... })
>>> return render_to_response("some_template.html",
...                           {"obj": world_pay},
...                           context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration obj %}
```

Template renders to something like below:

```
<form method='post' action='https://select-test.wp3.rbsworldpay.com/wcc/purchase'>
  <input type="hidden" name="futurePayType" id="id_futurePayType" />
  <input type="hidden" name="intervalUnit" id="id_intervalUnit" />
  <input type="hidden" name="intervalMult" id="id_intervalMult" />
  <input type="hidden" name="option" id="id_option" />
  <input type="hidden" name="noOfPayments" id="id_noOfPayments" />
  <input type="hidden" name="normalAmount" id="id_normalAmount" />
  <input type="hidden" name="startDelayUnit" id="id_startDelayUnit" />
  <input type="hidden" name="startDelayMult" id="id_startDelayMult" />
  <input type="hidden" name="instId" value="WP_ID" id="id_instId" />
  <input type="hidden" name="cartId" value="TEST123" id="id_cartId" />
  <input type="hidden" name="amount" value="100" id="id_amount" />
  <input type="hidden" name="currency" value="USD" id="id_currency" />
  <input type="hidden" name="desc" value="Test Item" id="id_desc" />
  <input type="hidden" name="testMode" value="100" id="id_testMode" />
  <input type="hidden" name="signatureFields" value="instId:amount:cartId" id="id_signatureFields" />
  <input type="hidden" name="signature" value="6c165d7abea54bf6c1ce19af60359a59" id="id_signature" />
  <input type='submit' value='Pay through WorldPay'/>
</form>
```

CHAPTER 22

Amazon Flexible Payment Service

Amazon FPS, is a service that allows for building very flexible payment systems. The service can be classified as a part Gateway and part Integration (offsite processor). This is because the customer is redirected to the Amazon site where he authorizes the payment and after this the customer is redirected back to the merchant site with a token that is used by the merchant to transact with the customer. In plain offsite processors, the authorization and transaction take place in one shot almost simultaneously.

Since the service isn't conventional (though very flexible), implementing FPS in merchant takes a couple of steps more.

The documentation for the service is available at Amazon FPS Docs.

Note: This integration has a dependency on `boto`, a popular AWS library for python.

Settings attributes required for this integration are:

- `AWS_ACCESS_KEY`: The Amazon AWS access key available from the user's AWS dashboard.
- `AWS_SECRET_ACCESS_KEY`: The Amazon AWS secret access key also available from the user's dashboard. Shouldn't be distributed to anyone.

Settings attributes:

```
MERCHANT_TEST_MODE = True
MERCHANT_SETTINGS = {
    "amazon_fps": {
        "AWS_ACCESS_KEY": "???",
        "AWS_SECRET_ACCESS_KEY": "???"
    }
}
```

Here are the methods and attributes implemented on the `AmazonFpsIntegration` class:

- `__init__(options = {})`: The constructor takes a dictionary of options that are used to initialize the underlying `FPSConnection` that is bundled with `boto`.
- `service_url`: A property that returns the API Endpoint depending on whether the the integration is in `test_mode` or not.
- `link_url`: A property that returns the link which redirects the customer to the Amazon Payments site to authorize the transaction.
- `purchase(amount, options={})`: The method that charges a customer right away for the amount `amount` after receiving a successful token from Amazon. The `options` dictionary is generated from the `return_url` on successful redirect from the Amazon payments page. This method returns a dictionary

with two items, `status` representing the status and `response` representing the response as described by `boto.fps.response.FPSResponse`.

- `authorize(amount, options={})`: Similar to the `purchase` method except that it reserves the payment and doesn't not charge until a `capture` (settle) is not called. The response is the same as that of `purchase`.
- `capture(amount, options={})`: Captures funds from an authorized transaction. The response is the same as the above two methods.
- `credit(amount, options={})`: Refunds a part of full amount of the transaction.
- `void(identification, options={})`: Cancel/Null an authorized transaction.
- `fps_ipn_handler`: A method that handles the asynchronous HTTP POST request from the Amazon IPN and saves into the `AmazonFPSResponse` model.
- `fps_return_url`: This method verifies the source of the return URL from Amazon and directs to the transaction.
- `transaction`: This is the main method that charges/authorizes funds from the customer. This method has to be subclassed to implement the logic for the transaction on return from the Amazon Payments page.

22.1 Example

In any app that is present in the `settings.INSTALLED_APPS`, subclass the `AmazonFpsIntegration` and implement the `transaction` method. The file should be available under `<app>/integrations/<integration_name>_integration.py`:

```
class FpsIntegration(AmazonFpsIntegration):
    # The class name is based on the filename.
    # So if the files exists in <app>/integrations/fps_integration.py
    # then the class name should be FpsIntegration
    def transaction(self, request):
        # Logic to decide if the user should
        # be charged immediately or funds
        # authorized and then redirect the user
        # Below is an example:
        resp = self.purchase(10, {...})
        if resp["status"] == "Success":
           return HttpResponseRedirect("/success/")
        return HttpResponseRedirect("/failure/")
```

In urls.py:

```
from billing import get_integration
amazon_fps = get_integration("fps")
urlpatterns += patterns('',
  (r'^amazon_fps/', include(amazon_fps.urls)),
  # You'll have to register /amazon_fps/fps-notify-handler/ in the
  # Amazon FPS admin dashboard for the notification URL
)
```

In views.py:

```
from billing import get_integration
def productPage(request):
   amazon_fps = get_integration("fps")
   url_scheme = "http"
```

```
    if request.is_secure():
        url_scheme = "https"
    domain = RequestSite(request).domain
    fields = {"transactionAmount": "100",
              "pipelineName": "SingleUse",
              "paymentReason": "Merchant Test",
              "paymentPage": request.build_absolute_uri(),
              # Send the correct url where the redirect should happen
              "returnURL": "%s://%s%s" % (url_scheme,
                                          domain,
                                          reverse("fps_return_url")),
             }
     # You might want to save the fields["callerReference"] that
     # is auto-generated in the db or session to uniquely identify
     # this user (or use the user id as the callerReference) because
     # amazon passes this callerReference back in the return URL.
     amazon_fps.add_fields(fields)
     return render_to_response("some_template.html",
                               {"fps": amazon_fps},
                               context_instance=RequestContext(request))
```

In some_template.html:

```
{% load render_integration from billing_tags %}
{% render_integration fps %}
```

The above template renders the following code:

```
<p><a href="https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start?callerKey=AKIAI
```

Braintree Payments Transparent Redirect

Braintree Payments Transparent Redirect is a service offered by Braintree Payments to reduce the complexity of PCI compliance.

Note: This integration makes use of the official braintree python package offered by Braintree Payments. Please install it before you use this integration.

Refer to the Braintree Payments Server to Server Gateway for the settings attributes.

Here are the methods and attributes implemented on the `BraintreePaymentsIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the Braintree environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- `service_url(self)`: A property that provides the URL to which the Transparent Redirect form is submitted.
- `get_urls(self)`: The method sets the url to which Braintree redirects after the form submission is successful. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

braintree = get_integration("braintree_payments")

urlpatterns += patterns('',
    (r'^braintree/', include(braintree.urls)),
)
```

- `braintree_notify_handler(self, request)`: The view method that handles the confirmation of the transaction after successful redirection from Braintree.
- `braintree_success_handler(self, request, response)`: If the transaction is successful, the `braintree_notify_handler` calls the `braintree_success_handler` which renders the `billing/braintree_success.html` with the `response` object. The `response` object is a standard braintree result described here.
- `braintree_failure_handler(self, request, response)`: If the transaction fails, the `braintree_notify_handler` calls the `braintree_failure_handler` which renders the `billing/braintree_error.html` with the `response` which is a standar braintree error object.
- `generate_tr_data(self)`: The method that calculates the tr_data to prevent a form from being tampered post-submission.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.braintree_payments_form`) and populates the initial data with the `self.fields`

(added through either the `add_fields` or `add_field` methods) and `tr_data`.

23.1 Example:

In the views.py:

```
braintree_obj = get_integration("braintree_payments")
# Standard braintree fields
fields = {"transaction": {
            "order_id": "some_unique_id",
            "type": "sale",
            "options": {
                "submit_for_settlement": True
              },
            },
            "site": "%s://%s" %("https" if request.is_secure() else "http",
                                RequestSite(request).domain)
        }
braintree_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"bp": braintree_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
braintree_obj = get_integration("braintree_payments")
urlpatterns += patterns('',
   (r'^braintree/', include(braintree.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration bp %}
```

CHAPTER 24

Stripe Payment Integration

Stripe Payment Integration is a service offered by Stripe Payment to reduce the complexity of PCI compliance.

Note: This integration makes use of the official stripe python package offered by Stripe Payments. Please install it before you use this integration.

Refer to the Stripe Payments Gateway for the settings attributes.

Here are the methods and attributes implemented on the `StripeIntegration` class:

- `__init__(self, options=None)`: The constructor method that configures the stripe setting
- `get_urls(self)`: The method sets the url to which the token is sent after the it is obtained from Stripe. This method is generally mapped directly in the `urls.py`.

```
from billing import get_integration

stripe_obj = get_integration("stripe")

urlpatterns += patterns('',
    (r'^stripe/', include(stripe_obj.urls)),
)
```

- `transaction(self, request)`: The method that receives the Stripe Token after successfully validating with the Stripe servers. Needs to be subclassed to include the token transaction logic.
- `generate_form(self)`: The method that generates and returns the form (present in `billing.forms.stripe_form`)

24.1 Example:

In <some_app>/integrations/stripe_example_integration.py:

```
from billing.integrations.stripe_integration import StripeIntegration

class StripeExampleIntegration(StripeIntegration):
    def transaction(self, request):
        # The token is received in the POST request
        resp = self.gateway.purchase(100, request.POST["stripeToken"])
        if resp["status"] == "SUCCESS":
            # Redirect if the transaction is successful
            ...
        else:
```

```
                    # Transaction failed
                    ...
```

In the views.py:

```
    stripe_obj = get_integration("stripe_example")
    return render_to_response("some_template.html",
                              {"stripe_obj": stripe_obj},
                               context_instance=RequestContext(request))
```

In the urls.py:

```
stripe_obj = get_integration("stripe_example")
urlpatterns += patterns('',
   (r'^stripe/', include(stripe_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration stripe_obj %}
```

eWAY Payment Integration

The eWAY integration functionality interfaces with eWAY's Merchant Hosted Payments facility. Their service makes it extremely easy to be PCI-DSS compliant by allowing you to never receive customer credit card information.

Note: This integration requires the suds package. Please install it before you use this integration.

The basic data flow is as follows:

1. Request an *access code* from eWAY.
2. Create an HTML form with the *access code* and user credit card fields.
3. Encourage the user to submit the form to eWAY and they'll be redirected back to your site.
4. Use the *access code* to ask eWAY if the transaction was successful.

You must add the following to project's settings:

```
MERCHANT_SETTINGS = {
    "eway": {
        "CUSTOMER_ID": "???",
        "USERNAME": "???",
        "PASSWORD": "???",
    }
}
```

The integration class is used to request an *access code* and also to check its success after the redirect:

class **EwayIntegration**(*access_code=None*)

Creates an integration object for use with eWAY.

access_code is optional, but must be configured prior to using *check_transaction()*.

request_access_code(*payment*, *redirect_url*, *customer=None*, *billing_country=None*, *ip_address=None*)

Requests an *access code* from eWAY to use with a transaction.

Parameters

- **payment** (*dict*) – Information about the payment
- **redirect_url** (*unicode*) – URL to redirect the user to after payment
- **customer** (*dict*) – Customer related information
- **billing_country** (*unicode alpha-2 country code (as per ISO 3166)*) – Customer's billing country

- **ip_address** (*unicode*) – Customer's IP address

Returns (access_code, customer)

The integration is automatically updated with the returned access code.

Supported keys in `customer`:

Key	Notes
`token_customer_id`	
`save_token`	
`reference`	
`title`	required for `save_token`
`first_name`	required for `save_token`
`last_name`	required for `save_token`
`company_name`	
`job_description`	
`street`	
`city`	
`state`	
`postal_code`	
`country`	required for `save_token`
`email`	
`phone`	
`mobile`	
`comments`	
`fax`	
`url`	

Supported keys in `payment`:

Key	Notes
`total_amount`	*required* (**must be cents**)
`invoice_number`	
`invoice_description`	
`invoice_reference`	

To add extra security, it's a good idea to specify `ip_address`. The value is given to eWAY to allow them to ensure that the POST request they receive comes from the given address. E.g.:

```
def payment(request):
    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(..., ip_address=request.META["REMOTE
    # ...
```

Returned value

The returned value is a tuple `(access_code, customer)`. `access_code` is the access code granted by eWAY that must be included in the HTML form, and is used to request transaction status after the redirect.

`customer` is a dict containing information about the customer. This is particularly useful if you make use of `save_token` and `token_customer_id` to save customer details on eWAY's servers. Keys in the dict are:

- `token_customer_id`
- `save_token`
- `reference`
- `title`

- first_name
- last_name
- company_name
- job_description
- street
- city
- state
- postal_code
- country – e.g. au
- email
- phone
- mobile
- comments
- fax
- url
- card_number – e.g. 444433XXXXXX1111
- card_name
- card_expiry_month
- card_expiry_year

check_transaction()

Check with eWAY what happened with a transaction.

This method requires access_code has been configured.

Returns dict

Key	Example
access_code	
authorisation_code	"198333"
response_code	"00"
response_message	"Transaction Approved" or None
option_1	"a1b2c3"
option_2	
option_3	
invoice_number	"19832261"
invoice_reference	"19832261-AA12/1"
total_amount	"1000"
transaction_id	"7654321"
transaction_status	True
error_message	
token_customer_id	"1234567890123456"
beagle_score	10.23

25.1 Example:

```
# views.py
from billing import get_integration
from django.shortcuts import get_object_or_404


def payment(request, cart_pk):
    # Pretend some 'Order' model exists with a 'total_price' in dollars
    order = get_object_or_404(Order, pk=cart_pk)

    integration = get_integration("eway_au")
    access_code, customer = integration.request_access_code(
        customer={"first_name": "Bradley", "last_name": "Ayers"},
        payment={"total_amount": order.total_price * 100},
        return_url=reverse(payment_done))
    request.session["eway_access_code"] = integration.access_code
    return render(request, "payment.html", {"integration": integration})


def payment_done(request, cart_pk):
    order = get_object_or_404(Order, pk=cart_pk)
    access_code = request.session["access_code"]
    integration = get_integration("eway_au", access_code=access_code)
    # Retrieve transaction status from eWAY
    status = integration.check_transaction()
    if status["response_code"] in ("00", "08", "11"):
        order.is_paid = True
        order.save()
        template = "receipt.html"
    else:
        template = "payment_failed.html"
    return render(request, template, {"status": status})
```

In order for eWAY to process the transaction, the user must submit the payment HTML form directly to eWAY. The helper tag `{% eway %}` makes this trivial:

```
{% load render_integration from billing_tags %}
{% render_integration integration %}
```

For a more configurable form, use the following pattern:

```
<form method="post" action="{{ integration.service_url }}">
    {{ integration.generate_form.as_p }}
    <input type="submit"/>
</form>
```

CHAPTER 26

Authorize.Net Direct Post Method

Authorize.Net Direct Post Method is a service offered by Authorize.Net to reduce the complexity of PCI compliance.

Here are the following settings attributes that are required:

- `LOGIN_ID`: The Login id provided by Authorize.Net. Can be obtained from the dashboard.
- `TRANSACTION_KEY`: The Transaction key is used to sign the generated form with a shared key to validate against form tampering.
- `MD5_HASH`: This attribute is used to generate a hash that is verified against the hash sent by Authorize.Net to confirm the request's source.

Here are the methods and attributes implemented on the `AuthorizeNetDpmIntegration` class:

- `__init__(self)`: The constructor that configures the Authorize.Net Integration environment setting it either to production or sandbox mode based on the value of `settings.MERCHANT_TEST_MODE`.
- **`form_class(self)`: Returns the form class that is used to generate the form.** Defaults to `billing.forms.authorize_net_forms.AuthorizeNetDPMForm`.
- `generate_form(self)`: Renders the form and generates some precomputed field values.
- `service_url(self)`: Returns the Authorize.net url to be set on the form.
- `verify_response(self, request)`: Verifies if the relay response originated from Authorize.Net.
- `get_urls(self)`: The method sets the url to which Authorize.Net sends a relay response, redirects on a success or failure.

```
from billing import get_integration

integration = get_integration("authorize_net_dpm")

urlpatterns += patterns('',
   (r'^authorize_net/', include(integration.urls)),
)
```

- `authorize_net_notify_handler(self, request)`: The view method that handles the verification of the response, firing of the signal and sends out the redirect snippet to Authorize.Net.
- `authorize_net_success_handler(self, request)`: The method that renders the *billing/authorize_net_success.html*.
- `authorize_net_failure_handler(self, request)`: The method that renders the *billing/authorize_net_failure.html*.

26.1 Example:

In the views.py:

```
int_obj = get_integration("authorize_net_dpm")
fields = {'x_amount': 1,
          'x_fp_sequence': datetime.datetime.now().strftime('%Y%m%d%H%M%S'),
          'x_fp_timestamp': datetime.datetime.utcnow().strftime('%s'),
          'x_recurring_bill': 'F',
         }
int_obj.add_fields(fields)
return render_to_response("some_template.html",
                          {"adp": int_obj},
                          context_instance=RequestContext(request))
```

In the urls.py:

```
int_obj = get_integration("authorize_net_dpm")
urlpatterns += patterns('',
   (r'^authorize_net/', include(int_obj.urls)),
)
```

In the template:

```
{% load render_integration from billing_tags %}
{% render_integration adp %}
```

CHAPTER 27

Signals

The signals emitted by Merchant are:

- *transaction_was_successful(sender, type=..., response=...)*: This signal is dispatched when a payment is successfully transacted. The *sender* is the object which has dispatched the signal. *type* is the kind of transaction. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success. Please consult the individual gateway docs for the response object.
- *transaction_was_unsuccessful(sender, type=..., response=...)*: This signal is dispatched when a payment fails. The *sender* is the object which has dispatched the signal. *type* is the kind of transation. Current choices for type are:
  - *purchase*
  - *authorize*
  - *capture*
  - *credit*
  - *void*
  - *store*
  - *unstore*

  response is the actual response object that is sent after the success.

  Note: Some gateways are implemented to raise an error on failure. This exception may be passed as the response object. Please consult the docs to confirm.

Writing a new gateway

Writing a new gateway for Merchant is very easy. Here are the steps to follow to write a new gateway:

- Create a new gateway file under the *billing.gateways* module which should follow this naming convention:

```
<gateway_name>_gateway.py
```

  So for example, PayPal would have *pay_pal_gateway.py*. Similarly, Authorize.Net, would have *authorize_net_gateway.py*.

- Create a class in this file with the following name:

```
class GatewayNameGateway(Gateway):
...
```

  So for PayPal, it would be *PayPalGateway* and for Authorize.Net, it would be *AuthorizeNetGateway*.

- Implement all or any of following methods in the class:

```
def purchase(self, money, credit_card, options = None):
...

def authorize(self, money, credit_card, options = None):
...

def capture(self, money, authorization, options = None):
...

def void(self, identification, options = None):
...

def credit(self, money, identification, options = None):
...

def recurring(self, money, creditcard, options = None):
...

def store(self, creditcard, options = None):
...

def unstore(self, identification, options = None):
...
```

Customizing Merchant

While we make all attempts to cover most of the functionality of the payment processors but may fall short sometimes. There is absolutely no need to worry as the gateway and integration objects are extensible.

Merchant looks for gateways and integration objects under every `INSTALLED_APPS` in `settings.py`. So it is possible for you to write your custom or modified objects within your app without having to patch the merchant code.

Note: Most of what is written below will also be applicable for gateways and you will have to replace instances of `integration` with `gateway`.

Suppose you want to extend the Braintree Payments Integration, to render a different template on success instead of the default `billing/braintree_success.html`.

Here is the process:

- In any of the `settings.INSTALLED_APPS`, create an `integrations` module (in layman's term an `integrations` directory with an `__init__.py` file under that directory).
- Create a file in that `integrations` directory that follows the convention below:

```
<integration_name>_integration.py
```

  Let us name the modified integration as `modified_bp`, then the filename would be:

```
modified_bp_integration.py
```

  and the Integration class name in that file as `ModifiedBpIntegration`.

  Note: The naming of the file and class follows a simple rule. The filename is split on underscores and each element of the split sequence is capitalized to obtain the class name.

  So in our example, in the `modified_bp_integration.py`:

```
class ModifiedBpIntegration(BraintreePaymentsIntegration):
    def braintree_success_handler(self, request, response):
       return render_to_response("my_new_success.html",
                                 {"resp": response},
                                 context_instance=RequestContext(request))
```

- Then use the new integration in your code just as you would for a built-in integration:

```
>>> bp_obj = get_integration("modified_bp")
```

CHAPTER 30

Contributing to Merchant

While there is no requirement for you to contribute your new gateway code or changes back to the upstream project, you can play a good samaritan by contributing back to the project and helping scores of people.

Here are the steps to follow to contribute back to Merchant:

- Fork the project from it's github page.
- Make the changes in your fork.
- File an issue at the github page and enclose a pull request.

  Note: If you want to include a new gateway, we request you to include a few tests (probably using the current tests as a template).

- Don't forget to add yourself to the *CONTRIBUTORS.txt* file before opening an issue.

CHAPTER 31

Changes

31.1 0.4 (upcoming)

- Added python3 support
- Removed google checkout

31.2 0.3

- Django 1.7 suport (**potentially backwards incompatible changes**)
- Updated the example *requirements.txt*

31.3 0.2

- Unit tests are skipped unless the corresponding gateways are configured
- Bugfix - Use settings.AUTH_USER_MODEL instead of get_user_model
- Demo - fill up initial data for all gateways

31.4 0.1

- Added PIN payments support

31.5 0.09

- Removed Samurai gateway and integration

31.6 0.08

- Added bitcoin backend
- Bugfixes to eWay, paypal integration and authorize.net

- Google Checkout shipping, tax rate and private data support
- Changes to Amazon FPS to work with latest boto. Addition of new fields to the FPS response model. A backwards incompatible change
- Made merchant django v1.5 compatible
- Fixes in the chargebee gateway broken by changes in the 'requests' api
- Changes to the example to prevent empty forms from raising a Server Error

31.7 0.07

- Added Chargebee support
- Added Beanstream gateway

31.8 0.06

- Added WePay gateway
- Added Authorize.Net Direct Post Method integration

31.9 0.05

- Added Paylane gateway support.

31.10 0.04

- Backwards incompatible version.
- Changes in the settings attributes. Now there is a single attribute for storing the configuration of all gateways and integrations. Check the docs for details.
- Changed the usage of the template tags. Refer the docs for details.
- Added a display_name to the integration object. Shouldn't affect users.

31.11 0.03

- Added support for Stripe and Samurai gateways and integrations.

31.12 0.02

- Added a setup.py and uploaded the package to pypi

31.13 0.01

- Initial commit.
- Overview
- Install Merchant
- Credit Card
- Gateways
  - On-site Processing
    - Authorize.Net
    - Braintree Payments Server to Server
    - eWay
    - PayPal
    - Stripe Payments
    - Paylane
    - WePay
    - Beanstream
    - Chargebee
    - Bitcoin
    - `Global Iris`
  - Off-site Processing
    - PayPal
    - `Google Checkout`
    - RBS WorldPay
    - Amazon FPS
    - Braintree Payments Transparent Redirect
    - Stripe
    - eWAY
    - Authorize.Net Direct Post Method
    - `Global Iris RealMPI`
- Signals
- Writing your own gateway
- Customizing the inbuilt Gateway/Integrations
- Contributing to Merchant
- Changelist

CHAPTER 32

Indices and tables

- genindex
- modindex
- search

C

check_transaction() (built-in function), 42, 76, 109, 143, 177, 211, 244, 278, 312, 346, 379, 413, 447, 481, 514, 548, 582, 616, 649, 683, 717, 751, 784, 818, 852, 886, 919, 953, 987, 1021, 1054, 1088, 1122, 1156, 1189, 1223, 1257, 1291, 1324, 1358, 1392, 1426, 1459, 1493, 1527, 1561, 1594, 1628, 1662, 1696, 1729, 1763, 1797, 1831, 1864, 1898, 1932, 1966, 1999, 2033, 2067, 2101, 2134, 2168, 2202, 2236, 2269, 2303, 2337, 2371, 2404, 2438, 2472, 2506, 2539, 2573, 2607, 2641, 2674, 2708, 2742, 2776, 2809, 2843, 2877, 2911, 2944, 2978, 3012, 3046, 3079, 3113, 3147, 3181, 3214, 3248, 3282, 3316, 3349, 3383, 3417, 3451, 3484, 3518, 3552, 3586, 3619, 3653, 3687, 3721, 3754, 3788, 3822, 3856, 3889, 3923, 3957, 3991, 4024, 4058, 4092, 4126, 4159, 4193, 4227, 4261, 4294, 4328, 4362, 4396, 4429, 4463, 4497, 4531, 4564, 4598, 4632, 4666, 4699, 4733, 4767, 4801, 4834, 4868, 4902, 4936, 4969, 5003, 5037, 5071, 5104, 5138, 5172, 5206, 5239, 5273, 5307, 5341, 5374, 5408, 5442, 5476, 5509, 5543, 5577, 5611, 5644, 5678, 5712, 5746, 5779, 5813, 5847, 5881, 5914, 5948, 5982, 6016, 6049, 6083, 6117, 6151, 6184, 6218, 6252, 6286, 6319, 6353, 6387, 6421, 6454, 6488, 6522, 6556, 6589, 6623, 6657, 6691, 6724, 6758, 6792, 6826, 6859, 6893, 6927, 6961, 6994, 7028, 7062, 7096, 7129, 7163, 7197, 7231, 7264, 7298, 7332, 7366, 7399, 7433, 7467, 7501, 7534, 7568, 7602, 7636, 7669, 7703, 7737, 7771, 7804, 7838, 7872, 7906, 7939, 7973, 8007, 8041, 8074, 8108, 8142, 8176, 8209, 8243, 8277, 8311, 8344, 8378, 8412, 8446, 8479, 8513, 8547, 8581, 8614, 8648, 8682, 8716, 8749, 8783, 8817, 8851, 8884, 8919, 8954, 8990, 9026, 9091

E

EwayIntegration (built-in class), 40, 74, 107, 141, 175, 209, 242, 276, 310, 344, 377, 411, 445, 479, 512, 546, 580, 614, 647, 681, 715, 749, 782, 816, 850, 884, 917, 951, 985, 1019, 1052, 1086, 1120, 1154, 1187, 1221, 1255, 1289, 1322, 1356, 1390, 1424, 1457, 1491, 1525, 1559, 1592, 1626, 1660, 1694, 1727, 1761, 1795, 1829, 1862, 1896, 1930, 1964, 1997, 2031, 2065, 2099, 2132, 2166, 2200, 2234, 2267, 2301, 2335, 2369, 2402, 2436, 2470, 2504, 2537, 2571, 2605, 2639, 2672, 2706, 2740, 2774, 2807, 2841, 2875, 2909, 2942, 2976, 3010, 3044, 3077, 3111, 3145, 3179, 3212, 3246, 3280, 3314, 3347, 3381, 3415, 3449, 3482, 3516, 3550, 3584, 3617, 3651, 3685, 3719, 3752, 3786, 3820, 3854, 3887, 3921, 3955, 3989, 4022, 4056, 4090, 4124, 4157, 4191, 4225, 4259, 4292, 4326, 4360, 4394, 4427, 4461, 4495, 4529, 4562, 4596, 4630, 4664, 4697, 4731, 4765, 4799, 4832, 4866, 4900, 4934, 4967, 5001, 5035, 5069, 5102, 5136, 5170, 5204, 5237, 5271, 5305, 5339, 5372, 5406, 5440, 5474, 5507, 5541, 5575, 5609, 5642, 5676, 5710, 5744, 5777, 5811, 5845, 5879, 5912, 5946, 5980, 6014, 6047, 6081, 6115, 6149, 6182, 6216, 6250, 6284, 6317, 6351, 6385, 6419, 6452, 6486, 6520, 6554, 6587, 6621, 6655, 6689, 6722, 6756, 6790, 6824, 6857, 6891, 6925, 6959, 6992, 7026, 7060, 7094, 7127, 7161, 7195, 7229, 7262, 7296, 7330, 7364, 7397, 7431, 7465, 7499, 7532, 7566, 7600, 7634, 7667, 7701, 7735, 7769, 7802, 7836, 7870, 7904, 7937, 7971, 8005, 8039, 8072, 8106, 8140, 8174, 8207, 8241, 8275, 8309, 8342, 8376, 8410, 8444, 8477, 8511, 8545, 8579, 8612, 8646, 8680, 8714, 8747, 8781, 8815, 8849, 8882, 8917, 8952, 8988, 9024, 9089

R

request_access_code() (built-in function), 40, 74, 107, 141, 175, 209, 242, 276, 310, 344, 377, 411, 445, 479, 512, 546, 580, 614, 647, 681, 715, 749, 782, 816, 850, 884, 917, 951, 985, 1019, 1052, 1086, 1120, 1154, 1187, 1221, 1255, 1289, 1322, 1356, 1390, 1424, 1457, 1491, 1525, 1559, 1592, 1626, 1660, 1694, 1727, 1761, 1795, 1829, 1862, 1896, 1930, 1964, 1997, 2031, 2065, 2099, 2132, 2166, 2200, 2234, 2267, 2301, 2335, 2369, 2402, 2436, 2470, 2504, 2537, 2571, 2605, 2639, 2672, 2706, 2740, 2774, 2807, 2841, 2875, 2909, 2942, 2976, 3010, 3044, 3077, 3111, 3145, 3179, 3212, 3246, 3280, 3314, 3347, 3381, 3415, 3449, 3482, 3516, 3550, 3584, 3617, 3651, 3685, 3719, 3752, 3786, 3820, 3854, 3887, 3921, 3955, 3989, 4022, 4056, 4090, 4124, 4157, 4191, 4225, 4259, 4292, 4326, 4360, 4394, 4427, 4461, 4495, 4529, 4562, 4596, 4630, 4664, 4697, 4731, 4765, 4799, 4832, 4866, 4900, 4934, 4967, 5001, 5035, 5069, 5102, 5136, 5170, 5204, 5237, 5271, 5305, 5339, 5372, 5406, 5440, 5474, 5507, 5541, 5575, 5609, 5642, 5676, 5710, 5744, 5777, 5811, 5845, 5879, 5912, 5946, 5980, 6014, 6047, 6081, 6115, 6149, 6182, 6216, 6250, 6284, 6317, 6351, 6385, 6419, 6452, 6486, 6520, 6554, 6587, 6621, 6655, 6689, 6722, 6756, 6790, 6824, 6857, 6891, 6925, 6959, 6992, 7026, 7060, 7094, 7127, 7161, 7195, 7229, 7262, 7296, 7330, 7364, 7397, 7431, 7465, 7499, 7532, 7566, 7600, 7634, 7667, 7701, 7735, 7769, 7802, 7836, 7870, 7904, 7937, 7971, 8005, 8039, 8072, 8106, 8140, 8174, 8207, 8241, 8275, 8309, 8342, 8376, 8410, 8444, 8477, 8511, 8545, 8579, 8612, 8646, 8680, 8714, 8747, 8781, 8815, 8849, 8882, 8917, 8953, 8988, 9024, 9089